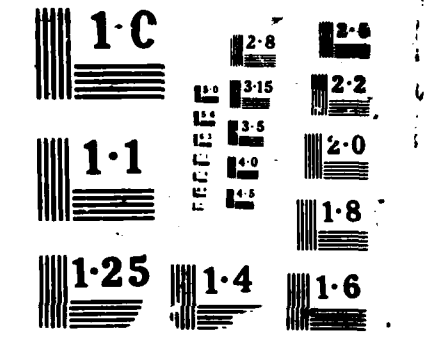
AD-A188 142 IMPLEMENTING DYNAMIC ARRAYS: A CHALLENGE FOR HIGH-PERFORMANCE MACHINES(U) NORTH CAROLINA UNIV AT CHAPEL HILL DEPT OF COMPUTER SCIENCE G MAGO ET AL 1/1

UNCLASSIFIED 1986 DAAL03-86-G-0050 F/G 12/6 NL


1·C
1·1
1·25
1·4
2·8
3·15
3·5
4·0
2·2
2·0
1·8
1·6

Grant DAAL03-86-G-0050

Implementing Dynamic Arrays: A Challenge For High-Performance Machines

(Extended Abstract)

Gyula Magó and Will Partain

Department of Computer Science
University of North Carolina at Chapel Hill

Introduction

There is an increasing need for *high-performance* AI machines. What is unusual about AI is that its programs are typically *dynamic* in the way their execution unfolds and in the data structures they use. AI therefore needs machines that are *late-binding*.

Multiprocessors are often held out as the answer to AI's computing requirements. However, most success with multiprocessing has come from exploiting numerical computations' basic data structure—the static array (as in FORTRAN). A static array's structure does not change, so its elements (and the processing on them) may be readily distributed.

In AI, the ability to change and manipulate the structure of data is paramount; hence, the pre-eminence of the LISP list. Unfortunately, the traditional pointer-based list has serious drawbacks for distributed processing.

The *dynamic array* is a data structure that allows random access to its elements (like static arrays) yet whose structure—size and dimensions—can be easily changed, i.e., bound and re-bound at run-time. It combines the flexibility that AI requires with the potential for high performance through parallel operation.

A machine's implementation of dynamic arrays gives a good insight into its potential usefulness for AI applications. Therefore, here we outline the implementation of dynamic arrays on a machine we are developing. We hope this description will enable and encourage comparison with other proposed machines.

Dynamic arrays

A *static array*, as in FORTRAN, is a data structure that allows efficient random access to its elements; however, adding and deleting elements is not possible. A *list*, as in LISP, allows easy addition and deletion of elements (on its front), but it does not offer random access to its elements. A *queue* (a doubly-linked list) allows easy addition and deletion of elements on both its ends, but it still does not provide random access.

A *dynamic array* (or *random access list*) provides easy addition and deletion of elements anywhere while still providing efficient random access to its elements. Dynamic arrays may sound too good to be true (and they apparently cannot be implemented effectively on conventional computers); nonetheless, they are the fundamental data type on the FFP machine.

The FFP machine's implementation of dynamic arrays

The FFP machine project at UNC-CH has as its goal the design and construction of a general-purpose computer equally adept at symbolic and numeric computation, whose high performance is balanced by its program-

ming flexibility. The successful implementation of dynamic arrays (called *sequences* in the FFP language) is crucial for achieving our project goals.

The FFP machine is a linear array of small processor/memory elements called *L cells* that communicate through interconnection networks built with *T cells*.

The FFP machine is a *small-grain multiprocessor*; a small-grain machine is one in which an individual processor cannot do useful computation by itself. A small-grain computer is the same thing as a *smart memory* (or a "logic-in-memory" system). Processor and memory are integrated; the separation of the two—the cause of the von Neumann bottleneck—is abolished. Programs are put into the memory, the memory "executes" them, and the results are removed from memory. Processing, as we traditionally imagine it, is completely distributed through the memory.

The key to implementing dynamic arrays in the FFP machine is the *addressless* nature of its smart memory—data is never bound to specific physical locations, so it can be relocated easily. This memory is between an addressable and an associative memory. In an addressable memory, locations are identified with their physical addresses; in an associative ("content-addressable") memory, all locations are fully interchangeable—data in such memory must be fully "self-describing". The FFP machine uses addresses in a limited way: its memory locations (L cells) have a built-in left-to-right order, and that order must be maintained if memory contents are relocated. Much of the structural information about the data is in the left-to-right order of the memory cells.

The FFP machine computes by repeatedly partitioning its hardware into submachines, one per reducible expression (both program and data), and letting the submachines do their reductions independently and simultaneously.

Program and data are stored in the FFP machine, one symbol per L cell; structure brackets are explicitly represented. A dynamic array is, therefore, spread out across many cells, occupying a segment of the L array.

A submachine begins reduction of an expression by building explicit structure descriptors ('directories') for its symbols; it keeps these directories until its reduction step is done. During that time, the data in each memory cell is self-describing, allowing easy access to elements of structures; thus, dynamic arrays' requirement for efficient random access is fulfilled.

As one reduction step gives way to the next, the directories describing the arrays' structure are re-calculated "on the fly"; thus, any change to a dynamic array's structure is figured into the directories on the next reduction step. In this way, dynamic arrays' requirement for efficient, uninhibited re-organization is met.

Storage management

One benefit of the chosen (nearly) addressless memory to support dynamic arrays in the FFP machine is that storage management can be integrated into the machine's hardware. Besides taking advantage of hardware-level speed, it also takes advantage of hardware-level parallelism. Fast storage management is of the greatest importance in a machine hoping to support dynamic data types, i.e., those whose structure can change at run-time.

Storage management in hardware is possible because the FFP machine's memory contents can be freely moved left or right, provided their order is maintained. Therefore, a system of rapid, *machine-wide* storage management is simply a solution to the one-dimensional problem of how to move data left and right, to make room where needed and to swallow up room where available.

When one tries to take advantage of unbounded parallelism on a multiprocessor with finite resources, deadlock is a potential problem. The problem is worse in systems where programs and data tend to be fixed in particular memory locations. In the FFP machine, L cells' contents are relatively free to move. Excess computations can even overflow out of the L array into auxiliary memory, allowing the machine to continue with its computation. The overflow programs are re-loaded as space becomes available.

Another factor to consider is that storage management in the FFP machine (and all other 'operating system' functions built into the hardware) is an integral part of the machine operation, instead of a background process that pops up now and again. Therefore, it can be readily *accounted for* in the analysis of machine performance. This is important because the cost of storage management is significant for late-binding computers.

Designing storage management into the hardware is especially important for an AI-oriented parallel computer. The dynamic, unpredictable nature of AI computation requires ongoing quick adaptation as execution unfolds.

Summary

The implementation of dynamic arrays in the FFP machine is indicative of the innovation required to design multiprocessor systems suitable for AI and other applications that require late binding.

The features of the FFP machine that we have described do not guarantee that it will be a successful computer. Other key questions about the FFP machine are how effectively it can be built, e.g., the precise complexity of its working hardware components, and how it will behave under realistic, dynamic computing loads. Only a hardware prototype will answer these questions; our current work is directed to that end.

Acknowledgement

This work was supported in part by the U.S. Army Research Office under Grant DAAL03-86-G-0050.

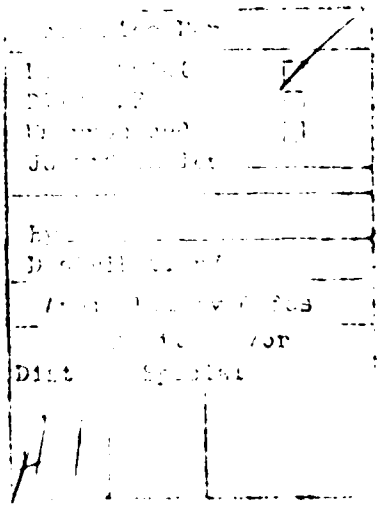
References

[1] Dybvig, K. R. Sequential and parallel architectures for functional languages. Ph.D. dissertation in preparation, University of North Carolina at Chapel Hill.

[2] Magó, G. A. Making Parallel Computation Simple: The FFP Machine. *IEEE Spring Compcon 1985*, pp. 424–428.

[3] Smith, B. Logic programming on an FFP machine. *Proceedings of the 1984 International Symposium on Logic Programming* (Febr. 6–9, 1984, Atlantic City, New Jersey), pp. 177–186.

[4] Wah, B. and Li, G.-J. *Computers for artificial intelligence applications*. IEEE Computer Society Press, 1986.



END

DATE

FILMED

3-88

DTIC